Graphfire: Synergizing Fetch, Insertion, and Replacement Policies for Graph Analytics

Aninda Manocha, *Student Member, IEEE*, Juan L. Aragón, *Member, IEEE*, and Margaret Martonosi, *Fellow, IEEE*

Abstract—Despite their ubiquity in many important big-data applications, graph analytic kernels continue to challenge modern memory hierarchies due to their frequent, long-latency, pointer indirect accesses to vertex property data. Such accesses exhibit poor locality and variable reuse that trouble cache replacement policies, and consequently increase memory bandwidth pressure. Specialized graph-tailored prefetching mechanisms, processor designs, and memory hierarchy engines have been developed to tolerate the long latencies of such accesses. However, these approaches are either too bandwidth-intensive, require invasive hardware changes that inhibit general-purpose computation flexibility, or rely on software preprocessing that limits true speedup.
This work introduces Graphfire, a flexible memory hierarchy approach that learns different access patterns in graph processing and exploits the synergy of specialized fetch, insertion, and replacement optimizations for problematic indirect accesses without relying on software or ISA support. More specifically, Graphfire identifies when these irregular accesses occur and employs tailored access granularities, data-aware insertion, and frequency-based replacement accordingly. It achieves up to a 1.79× speedup (geomean 1.3×) and these improvements scale due to bandwidth efficiency; with 64 cores, Graphfire yields up to a 71.33× speedup (geomean 63.32×) over a single baseline core and allows memory-bound graph analytic codes to scale far beyond prior work.

Index Terms—graph analytics, cache, memory hierarchy.

1 Introduction

For decades, caches have played a significant role in improving CPU performance, reducing off-chip memory access latency as well as processor-memory traffic [16], [43]. Many widespread applications, including linear algebra routines for dense neural networks, demonstrate access regularity that benefits from the logic and structure of modern cache designs. However, graph analytics remain an important domain of applications where even state-of-the-art cache management techniques continue to struggle.

Graph applications are gaining importance for machine learning and data analytics [13], [41]. Many kernels in this domain have unpredictable memory access patterns that arise from pointer indirection [5]. These irregular memory accesses correspond to the structure of the graph input, as it is traversed on a per-vertex basis in order for the application to gather and compute data about the graph. Unfortunately, such irregular accesses have very little temporal and spatial locality. Modern datasets, e.g. social networks, are also significantly larger than the last-level cache (LLC), leading to thrashing at all levels of the memory hierarchy. Lastly, the irregular accesses themselves have variable reuse, which troubles heuristic- and learning-based replacement policies that rely on recency or temporal access sequences. As a result, graph applications frequently perform expensive, off-chip memory accesses, whose long latencies can dominate application runtimes and limit scalability.

- *Aninda Manocha is with the Department of Computer Science, Princeton University, Princeton, NJ, 08540. Email: amanocha@princeton.edu.*
- *Juan L. Aragón is with the Computer Engineering Department, University of Murcia, Murcia, Spain, 30100. Email: jlaragon@um.es.*
- *Margaret Martonosi is with the Department of Computer Science, Princeton University, Princeton, NJ, 08540. Email: mrm@princeton.edu.*

When irregular access patterns are coupled with massive networks, software-only optimizations are unlikely to succeed efficiently. Instead, performance optimizations must utilize hardware to reduce long-latency DRAM accesses. A large body of work has spanned cache management techniques [17], [18], [39]. These approaches range from designing heuristics to capture cache-friendly and cache-averse accesses, to predicting reuse and re-reference intervals, dead blocks, and per-workload access patterns. However, none of these techniques consider specific access patterns of graph applications and instead focus on those that are well-known and common in more regular workloads, e.g. streaming, strided, thrashing, mixed, etc. Thus, any hardware overheads these techniques incur are wasted when they are applied to graph analytics.

GRASP [15] proposed the first step towards domain-specialized LLC management for graph analytics, but incurs software preprocessing costs for degree-based graph reordering [14]. Software preprocessing renders the technique less practical for large graphs, e.g. in many application scenarios where the input graph is only processed once [5], or when the graph is not even fully traversed, e.g. in search algorithms. To our knowledge, there exists no memory hierarchy approach for graph applications that learns and optimizes for their access patterns with *no* software support.

With the goal of optimizing cache performance for graph applications, our work makes the following key observations: (i) The memory hierarchy must specialize for the *problematic indirect accesses (PIAs)* to alleviate their bottlenecks. (ii) To be software-agnostic, a lightweight mechanism must automatically identify PIAs, which can be achieved on a per-instruction basis. (iii) While PIAs are irregular, a subset of them have high reuse, so the LLC must retain them.

Our Approach: Given these observations, this paper presents Graphfire[1], a flexible, hardware-based memory hierarchy approach that (i) learns when PIAs occur in graph applications and (ii) optimizes their performance through tailored fetch, insertion, and replacement policies. Through an adaptive hardware locality predictor that monitors L1 cache accesses, Graphfire learns different access patterns on a per-PC basis and classifies them to identify which instruction(s) perform the PIAs. With this knowledge, Graphfire decides when to exploit the synergies between caching policies optimized for the PIAs. More specifically, these data-aware policies address issues in three main thrusts:

Fetch: All primary loads in graph applications are either streaming or pointer indirect, which presents a challenge for the memory hierarchy as they have different cacheline size needs. Fetching PIAs at *word*–rather than line–granularity improves cache utilization while allowing streaming accesses to continue benefiting from locality.

Insertion: Streaming contiguous accesses have no reuse once they fill a cacheline and thus require little cache space. Having these accesses bypass lower memory hierarchy levels devotes more cache space to high-reuse PIAs. This lowers their chances of being evicted from the LLC without harming the performance of more regular accesses.

Replacement: Despite their poor locality, a subset of PIAs have high reuse and benefit from caching [7], but their reuse distances are often too long to protect them from eviction. When the LLC is reserved for PIAs cached at word granularity, frequency-based replacement is significantly more effective at learning and retaining PIAs with high reuse, unlike across-the-board temporal locality.

Our work advances these F-I-R policies individually. More importantly, we are the first to note they need to be employed together for their full impact. Furthermore, they require only modest hardware additions and enable Graphfire to offer domain-specialized improvements while retaining excellent performance of other workloads. To summarize, this work's main contributions are:

1) We introduce a novel *hardware locality predictor* that learns graph application access patterns online and classifies them with no software or ISA support.
2) We develop a *flexible memory hierarchy design* that caters to PIAs by exploiting synergies between: (F) Tailored access granularities alleviate poor cache utilization. (I) Data-aware insertion reserves space for even more PIAs to fit in the LLC. (R) Frequency-based replacement leverages the additional LLC space to better learn and retain high-reuse PIAs.
3) We evaluate Graphfire on widely used workloads and achieve: (i) Single-core speedups up to $1.79\times$ (geomean $1.3\times$) on in-order and $1.6\times$ (geomean $1.2\times$) on OoO over state-of-the-art approaches. (ii) Scalable performance improvements through memory-level parallelism and reduced DRAM contention: $63.32\times$ speedup with 64 cores, much improved over the baseline speedup of $47.04\times$.

Overall, with nimble workload adaptation and lightweight hardware requirements, Graphfire offers important advances for graph analytics and data processing workloads.

1. Graphfire = ***Graph f***etch, ***i***nsertion, and ***re***placement.

```
1  while worklist not empty:
2    for v in worklist:
3      process_vertex(v)
4      start = vertex_ptr[v]
5      end = vertex_ptr[v+1]
6      for neib in neighbors[start:end]:
7        process_edge(neib)
8        neib_data = property_array[neib]
9        if update_neib(neib_data):
10         property_array[neib] = neib_data
11         updated = true;
12       if updated:
13         add_to_worklist(neib)
```

Fig. 1. Graph processing kernel example.

2 Motivation

2.1 Memory Access Patterns in Graph Processing

Graph applications are notorious for irregular memory accesses that arise from data traversals. State-of-the-art, work-efficient algorithm implementations perform graph traversals iteratively through two nested kernel loops and utilize the Compressed Sparse Row (CSR) format to efficiently store the input dataset as one-dimensional dense arrays [45]. Pointer indirect accesses occur in the *vertex property array*, which stores per-vertex results, e.g. distances or ranks [6]. CSR arrays store graph information, e.g. vertex and edge locations, but are accessed regularly and/or infrequently.

Fig. 1 presents pseudocode for graph processing kernels. The outer loop (lines 2-5) iterates through a worklist of vertices (for the current algorithm iteration) while the inner loop (lines 6-13) analyzes the current vertex's neighbors to potentially update their data, depending on the objective of the algorithm. If a neighbor is updated, it is added to the worklist for the next algorithm iteration (lines 12-13). The algorithm terminates when the worklist is empty (line 1), i.e. the graph has been traversed and updates have stabilized.

Memory access patterns in graph algorithms can be classified on a per-PC basis, assuming in-order processor execution. Instructions in the outer loop occur *infrequently* relative to those in the inner loop, particularly for graphs with high edge to vertex ratios. Accesses are either *streaming* or *indirect*. This yields four different access patterns:

Infrequent, Streaming: These accesses appear in the outer for loop or inside a conditional in the inner loop and have a streaming behavior. As line 2 of Fig. 1 shows, the algorithm iterates through a worklist of vertices and loads each vertex index in a streaming fashion. However, this access does not exhibit good locality because each access can be separated by several memory accesses in the inner loop of the kernel.

Infrequent, Indirect: These pointer indirect accesses appear in the outer for loop or a conditional in the inner loop. The current vertex `v` indexes into the `vertex_ptr` array to load its neighbor list indices (lines 4-5) and determine the number of inner loop iterations. The first load to the starting index (line 4) is pointer indirect, while the second (line 5) has locality. Another indirect access arises from conditionally updating vertex property data (line 10). However, these irregular accesses do not have a significant impact on performance, as they occur infrequently. We focus on the *primary* memory accesses, i.e. those that are executed in *every*

TABLE 1
Applications and inputs used in our evaluation and their properties.

Applications	Input	Type	Vertices	Edges	Avg / Max Deg.	Prop. Array Footprint % (BFS / SSSP / PRD / PR)
Breadth-First Search (BFS)	Kronecker (Kron)	synthetic, power-law	2.1M	64M	30 / 102440	1.55 / 1.38 / 1.48 / 3.00
Single Source Shortest Paths (SSSP)	LiveJournal (LiveJ)	real-world, social	4.8M	69M	14 / 20293	3.08 / 2.47 / 2.82 / 5.80
PageRank-Delta (PRD)	Orkut (Ork)	real-world, social	3.1M	117M	38 / 32998	1.25 / 1.13 / 1.20 / 2.43
PageRank (PR)	Pokec (Pok)	real-world, social	1.6M	30M	18 / 8763	2.41 / 2.02 / 2.25 / 4.60
	Wikipedia (Wiki)	real-world, web	1.8M	40M	21 / 6975	2.14 / 1.83 / 2.01 / 4.11

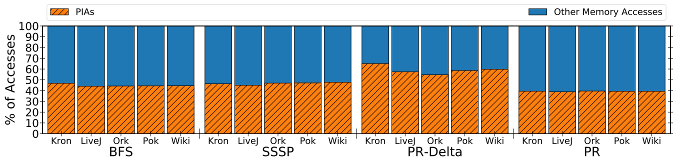


Fig. 2. PIAs (orange) are frequently accessed despite their small memory footprint relative to other data structures.

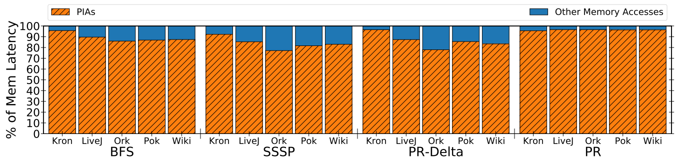


Fig. 3. PIAs (orange) dominate total memory access latency due to their frequency and irregularity.

inner loop iteration, as their access latencies significantly impact application runtime.

Primary Streaming Accesses (PSAs): These accesses occur in the critical path of the inner for loop and perform a streaming load or store to neighbor indices (line 6). These are true streaming accesses that exhibit both temporal and spatial locality and they occur frequently and regularly. Thus, they are cache-friendly and do not contribute to data supply bottlenecks. We refer to these accesses as *primary streaming accesses (PSAs)* to describe these characteristics.

Primary Indirect Accesses (PIAs): These accesses also occur in the critical path of the inner for loop and perform a pointer indirect load (highlighted in line 8) to the vertex property array in order to load data for a given neighbor. Because these indirect accesses occur in every loop iteration and have long latencies that incur performance costs, they are responsible for the data supply bottlenecks of graph applications. We refer to them as *primary indirect accesses (PIAs)* and detail their characteristics in the next section.

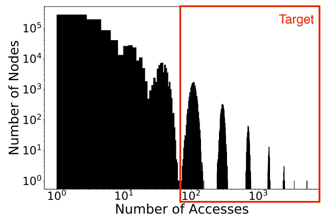
2.2 The Problems with PIAs

The *vertex property array*, the primary source of PIAs, comprises a very small percentage of the application's total memory footprint. The far right column of Tab. 1 presents percentages for different application/input combinations. These percentages depend on application data demands (worklist sizes) and input sizes. On average, the vertex property array is only 3.06% of the total memory footprint.

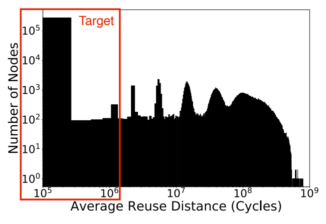
Fig. 2 shows that despite their relatively small data footprint, the PIAs themselves comprise a large fraction of the total number of accesses in the application. Frequent updates must be made to the vertex property array as the application traverses the graph. Combining *irregularity* with *frequency* yields a memory latency performance bottleneck.

Fig. 3 breaks down the total memory latency into PIAs vs. other memory accessses. Among all applications and inputs, the latencies of the PIAs average 88% of the total memory latency, comprising the main application performance bottleneck. Thus, innovations within the memory hierarchy are necessary to account for the lack of regularity and locality exhibited by these references.

Lack of Locality: Unfortunately, the irregularity of PIAs causes them to exhibit poor locality. On average, 54.1 (L1),



(a) Many PIAs have low access frequency; few have very high.



(b) PIA reuse distances vary significantly.

Fig. 4. Graphfire focuses on retaining vertices with high frequencies and low reuse distances (red).

59.9 (L2), and 59.5 (L3) bytes are *unused* in a 64B evicted cacheline for the application/input combinations above. Thus, fetching an entire cacheline's data for PIAs is wasteful.

Interference Between Access Patterns: The coexistence of different memory access patterns in the same cache set can hurt performance. PIAs can be evicted by PSAs or other infrequent accesses, leading to several conflict misses, particularly PIAs with high reuse that should be retained in the LLC [6]. PIAs are the primary eviction candidates in the LLC due to their irregularity and frequency. On average, 21% of their evictions are caused by other types of accesses. Removing this interference can improve the cache performance of PIAs.

Variable PIA Reuse: Fig. 4 presents PIA access and reuse histograms when BFS runs on a Kronecker network. This captures the power-law trend of many real-world datasets [22]. Fig. 4a shows that few vertices are accessed frequently, while most are accessed infrequently. Vertices with high reuse should not be evicted by low-reuse PIAs. Fig. 4b illustrates the variability in PIA reuse distance. Many are reused, but most reuse distances are too long (relative to cache associativities) to protect PIAs from eviction. Thus, the replacement policy for PIAs should adapt to vertex characteristics.

2.3 Tailoring Cache Management for Graph Analytics

Motivated by the preceding data, Graphfire (i) automatically classifies different access patterns on a per-instruction basis through online learning, (ii) identifies when problematic indirect accesses occur, and (iii) leverages synergistic cache optimizations to alleviate the performance bottlenecks these accesses create. These specializations within the memory hierarchy allow graph applications to achieve improved cache performance, and consequently significant overall speedups, on general-purpose hardware.

3 Learning Memory Access Patterns

3.1 Re-Reference Table for Online Locality Prediction

Graphfire learns different memory access patterns of graph applications on a per-PC basis in order to identify and specialize for PIAs. It accomplishes this through a decoupled hardware-based locality predictor that learns *online* by monitoring L1 cache accesses in parallel with cache operations. This predictor uses a re-reference table (RRT) to track reuse from spatial locality for each memory instruction's accesses. Each RRT entry contains four fields: (1) the PC, (2) a utilization value (UV), (3) the L1 cache set ID the PC most recently accessed, and (4) a frequency value. With each L1 access, the predictor is indexed by its PC and updates the RRT entry's UV, cache set ID, and frequency.

Cacheline Utilization: Each PC's UV is between 0 and N, where $N+1$ words fit in a cacheline ($N = 15$ with 64B cachelines). Upon an L1 hit, the predictor checks the cache set ID stored in the PC's entry. If it matches the cache set accessed, then the entry's UV increments. If it does not match or the access missed in the L1, then the UV decrements. After each access, the set ID is updated accordingly. A PC must access the same cacheline at least twice in a row to increase its UV. Thus, the UV signifies spatial locality; $UV = N$ indicates that the PC fully utilizes cachelines it accesses, while a $UV = 0$ indicates poor utilization.

Since graph applications exhibit primarily either streaming or indirect accesses, UVs quickly stabilize to values of 0 or N and the predictor uses these values to classify PCs as streaming or indirect accesses. If $UV > \frac{N+1}{2} - 1$, the PC has accessed at least half of the same cacheline contiguously (cache-friendly). Otherwise, the PC corresponds to either a pointer indirect or infrequent, streaming access pattern with long reuse distances.

PIA Identification: The RRT entry's frequency value is a saturating counter that increments with each PC access. If the value saturates, the PC corresponds to a primary memory access. Therefore, a PC that performs PIAs has a low UV and high frequency. To give time for UVs to stabilize and frequency values to accumulate, a global counter times a "learning" phase, during which no PIA cache optimizations are used. If this counter saturates and a PIA has been identified, Graphfire uses its predictor to apply cache optimizations on a per-PC basis. If no PIA has been identified, which may occur in more dense and regular applications, the counter resets to continue learning. The counter also resets when a new PC is inserted into the RRT, as a new part of the program has been reached. This adaptive learning is particularly useful for multi-phase and co-located applications, demonstrated in Sec. 7.

When a non-graph application is being executed, the RRT can be disabled entirely via a global on/off switch (determined by an OS or API call), which will prevent any memory access from being identified as a PIA. This effectively disables Graphfire, which can be useful when an application does not have irregular accesses with variable reuse, or the input graph has already been preprocessed and reordered. Overall, Graphfire has a flexible design.

3.2 RRT Performance

We evaluate the predictor's accuracy with graph kernels and multi-phase applications (Sec.6) running on five different networks (Tab.1). We measure accuracy by the percentage of PIA and PSA PCs learned correctly. The predictor learns primary accesses quickly, i.e. within 5-10 iterations of a kernel loop, and identifies both access types with 100% accuracy. Thus, Graphfire robustly learns per-PC memory access patterns based on their locality to identify the PIAs.

Hardware Overhead: Given a 32KB L1 cache with 64B blocks and 8-way set associativity, each RRT entry requires: (i) 64-bit PC, (ii) 4-bit RRV saturating counter, (iii) 8-bit cache set ID, and (iv) 3-bit frequency saturating counter. Therefore, each entry stores 10B. The number of entries necessary to capture the *primary* memory accesses depends on the workload. Large applications with many memory instructions can have a bounded RRT that only tracks the most frequently occurring PCs (PSAs and PIAs), while those not captured are labeled infrequent and irregular. Graph processing kernels have only a few PIA PCs. In our evaluated kernels and multi-phase applications, 32 entries is more than sufficient, leading to a 320B hardware overhead. Nevertheless, the RRT size can be increased, e.g. 2048 entries, to track more static loads and still occupy less than 1% of the area devoted to each core's caches.

4 Cache Policies for PIAs

With its knowledge of PIAs, Graphfire optimizes fetch, insertion, and replacement policies to mitigate long latencies.

4.1 Fetch: Tailored Access Granularity

Unlike streaming accesses, PIAs poorly utilize cachelines. Based on the observation that loads in graph applications are overwhelmingly streaming or indirect, Graphfire tailors access granularities accordingly. Streaming accesses exhibit high predictability and spatial locality that benefits from prefetching and large cachelines. On the other hand, PIAs are unpredictable and poorly utilize large cachelines, incurring many cold and capacity misses. Therefore, upon identification of a PIA, Graphfire fetches data at word granularity.

4.2 Insertion: Data-Aware Caching

Streaming accesses in graph applications have poor, if any, temporal locality once they contiguously fill a cacheline. PSAs load neighbor indices for the current vertex to process, which is often not processed again. Therefore, the L1 cache is sufficient for the PSAs. Contrarily, the PIAs' data footprint is significantly larger than the LLC capacity and interference from other accesses, e.g. streaming, create conflict. Despite their irregularity, a subset of the PIAs have high reuse that benefit from caching if more space is available for them.

Graphfire exploits PSAs' low cache capacity requirements and other non-PIAs' low frequencies to reserve as much space as possible, i.e. the L2 and L3, for the PIAs. Thus, all non-PIA accesses bypass these caches because they do not gain much benefit from them. This prevents other accesses from evicting high-reuse PIAs from the LLC.

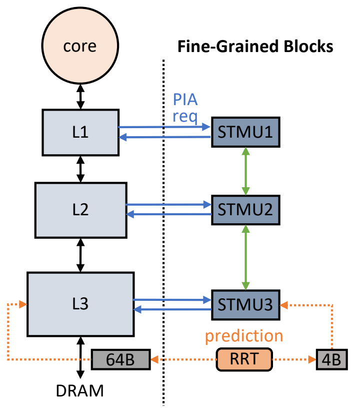


Fig. 5. Graphfire utilizes a STMU at each cache level to encapsulate fine-grained data operations (blue arrows). A cache sends a merged block of multiple sub-blocks to its STMU for any fine-grained data operation. The STMUs can also communicate fine-grained data with one another in the event of sub-block misses and evictions (green arrows). Upon insertion of new data in the hierarchy, the RRT predicts whether the memory access is streaming or irregular (PIAs), which decides whether to insert a normal (64B) block or sub-block (orange arrows).

4.3 Replacement: Frequency-Based Eviction

PIAs can be considered on a per-vertex basis during replacement. Real-world, power-law graphs have few high-degree vertices and many low-degree vertices (Fig. 4), creating many opportunities for low-degree PIAs to evict those with high degrees. Given the distribution of vertex degrees, high-reuse PIAs can be learned with access frequency. Frequency-based replacement (FBR) is usually not effective for modern caches because counters do not have enough time to accumulate before their corresponding cachelines are evicted. However, Graphfire's fetch and insertion techniques effectively increase the LLC capacity by improving cache utilization, creating significant opportunity for FBR to learn high-reuse PIAs. Thus, Graphfire applies FBR in the lower cache levels instead of relying on recency. Meanwhile, because the L1 primarily caches PSAs, LRU effectively evicts fully utilized cachelines of contiguous data.

5 Memory Hierarchy Design

To flexibly support fine-grained data fetches for PIAs in graph applications, Graphfire utilizes two types of cache blocks: (1) normal 64B cache blocks and (2) *merged blocks*, 64B of coalesced, non-contiguous *sub-blocks* that store data for fine-grained accesses. Sub-blocks can be 4B (integers or floats) or 8B (doubles or longs); each merged block has a granularity bit g that is set based on the memory instruction's data type. To perform operations on these specialized merged blocks, Graphfire introduces a *sub-tag matching unit (STMU)* for each cache to utilize. This mechanism employs *hierarchical tags* for fine-grained accesses to allow non-contiguous sub-blocks to share a cacheline without incurring *any* tag overhead.

Fig. 5 presents an overview of the memory hierarchy design. For fine-grained accesses, each cache interfaces with its STMU, which performs operations on merged blocks (blue arrows). The STMUs can also communicate with one another (green arrows). Upon insertion of new data from DRAM, the RRT predicts whether the memory access is streaming or irregular. The data is inserted as a normal (64B) block or a sub-block in a merged block (orange arrows).

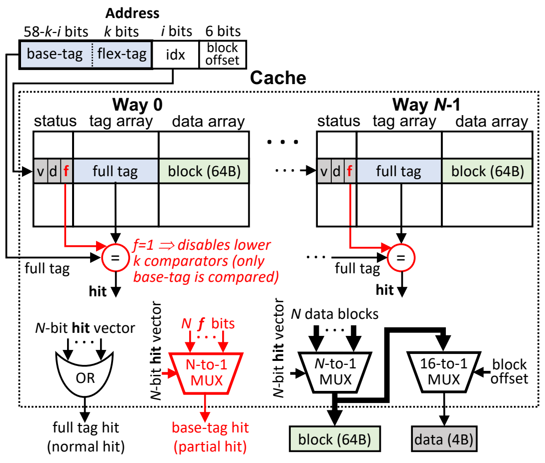


Fig. 6. The STMU requires modest cache modifications (red) to support merged blocks via hierarchical tags. Each cache block has an additional "f" bit to denote whether it is fine-grained or not and whether to use the base (lower k comparators disabled) or full tag for search. By default, this bit is set to 0 (normal cache block) and enabled only when a merged block (predicted by the RRT) is inserted into the cache. An additional multiplexer is used to raise a partial hit if the block is fine-grained ("f" bit enabled).

5.1 Hierarchical Tags with Sub-Tag Matching Units

To prevent sub-block metadata from imposing area overhead and requiring significant cache modifications, the STMU stores sub-block metadata *inside* the original data block. Fig. 6 illustrates the minimal cache modifications (red) necessary to support this design. The k least significant bits of the original tag make up the *flex-tag*, which combine with the block offset to create a *sub-tag* that identifies a particular sub-block stored in a merged block. The remaining bits of the original tag not used in the flex-tag make up the *base-tag*, which determines which cache block (way) a sub-block resides in for a given set. Therefore, all sub-blocks of a given merged block share the same base-tag (as opposed to the original tag) and are identified by their distinct sub-tags.

The cache performs first-level tag matching at block (e.g. 64B) granularity, and the STMU performs second-level tag matching at sub-block granularity. If the requested data is present in the cache, this results in either a *normal* (block access) or *partial* (sub-block access) hit, depending on whether the access was predicted as streaming or irregular. One fine-granularity bit f per cache block distinguishes fine- from coarse-grained accesses and is set by the RRT's prediction. This bit's value never changes for the remainder of the block's lifetime in the memory hierarchy; the block can experience either a normal *or* partial hit until it has been evicted from all caches. Thus, data for a given address can never be cached in a sub-block and block simultaneously. Note that the caches do not use PCs; only the RRT interfaces with the instruction cache.

The STMU operates on a block comprised of K data sub-blocks, each with its own sub-tag and status bit metadata (dirty/valid/coherency), which are necessary for sub-block matching. Because the metadata resides inside the original data block, some capacity is sacrificed. More specifically, a data block fits up to $blk_{size}/(subblk_{size} + subblk_{tag} + subblk_{metadata})$ sub-blocks as opposed to $blk_{size}/subblk_{size}$. Graphfire maintains the same base tag size across the cache hierarchy, so each cache level has a different number of flex-

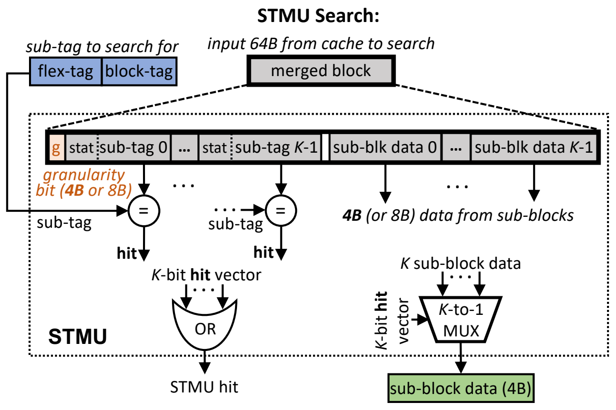


Fig. 7. To search for a sub-block within a merged block, the STMU operates on K sub-blocks in parallel. Each sub-block has its own sub-tag and metadata and the STMU compares the K sub-tags against the input sub-tag to determine if there is a sub-block hit, similar to a normal cache. Thus, the STMU outputs a hit signal and the 4B (or 8B) data requested (upon a load hit).

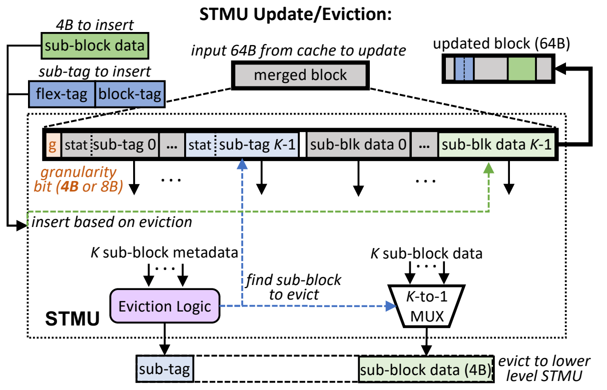


Fig. 8. To update a merged block with a new sub-block in the event of a sub-block miss, the STMU looks at the metadata of the K sub-blocks to select an eviction candidate, if the merged block is full, and replace it with the new sub-tag and data, or simply insert the new sub-block. The evicted sub-block is sent to the lower level cache if it is dirty.

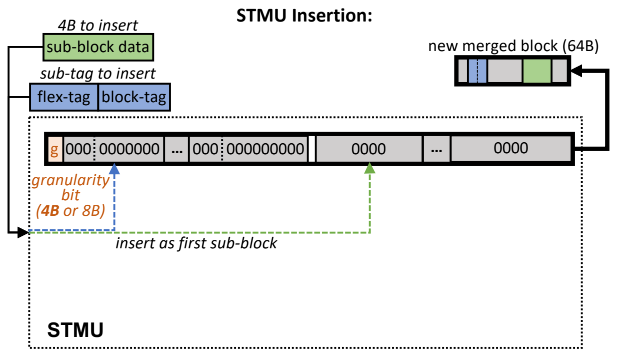


Fig. 9. To insert a sub-block as part of an entirely new merged block in the event of a merged block miss (base tag not found), the STMU resets all sub-block tags and data (to zeroes) and inserts the new sub-tag and data in the first respective locations.

tag bits, where the L1 (largest original tag) has the most and the LLC (smallest original tag) has the least. Despite this capacity sacrifice, our approach benefits PIAs that otherwise would waste most of the cacheline if they were accessed at block granularity; our results demonstrate this.

For example, a 2MB, 16-way LLC has 64B cache blocks (2048 sets). The original tag contains 47 bits (11 index bits and 6 block offset bits). If each sub-block has 4B of data and there are 2 flex-tag bits, then base tags have 45 bits and sub-tags jave $2+6=8$ bits. Therefore, $2^2 \times 2^{(6-2)} = 64$ sub-blocks can share the same base tag and match to the same merged block. Furthermore, each sub-block requires $32 + 8 + 10 = 50$ bits of storage (assuming 10 bits of metadata), allowing 10 4B sub-blocks to fit in a 64B block in the LLC.

5.2 STMU Operations

All fine-grained operations are forwarded to and encapsulated by each cache's STMU, so the original cache behavior remains unchanged. The STMU effectively acts as an encoder/decoder unit that intercepts 64B accesses between cache levels to access or insert 4B (or 8B) of data as a sub-block. To accomplish this, the STMU supports three operations, (1) *Search*, (2) *Update/Evict*, and (3) *Insert* when it receives a (64B) input merged block.

Search: Fig. 7 displays how the STMU searches for a sub-tag inside an input merged block (that comes from its corresponding cache) to check for a sub-block hit. The STMU compares K sub-tags to the input sub-tag in parallel, similar to how a normal cache performs a search, in order to determine if there is a sub-block hit or miss. In the case of a load hit, the 4B data requested is returned. For STMU1, this data is sent to the L1 cache to be delivered to the core, which allows the core to remain unmodified. For STMU2 and STMU3, the data is sent to the upper level STMU, i.e. STMU1 and STMU2, respectively. In the case of a store hit, the sub-block data corresponding to the located sub-tag is updated and the returned (stale) data is not used.

Update/Evict: Fig. 8 presents how the STMU updates a sub-block in a merged block. This occurs when there is a sub-block load miss and the sub-block data fetched from a lower level STMU needs to be inserted, or a store miss. If the merged block is full (all sub-block locations are occupied), the STMU looks at the K sub-blocks' metadata to identify an eviction candidate, using the Least Frequently Used policy, and replaces it with the new sub-block. Otherwise, the STMU simply inserts the new sub-block into an empty location. If there is an eviction candidate and it is dirty, then it is sent to the lower level STMU.

Insert: Fig. 9 shows how the STMU inserts a new merged block with a new sub-block in the event that there was a merge block miss, i.e. the base tag was not present in the cache. The STMU resets all sub-block tags and data to zeroes and inserts the new sub-block tag and data into the first sub-tag and data locations. This insertion of a merged block could potentially evict a normal (64B) block or another merged block. If the latter occurs, then the STMU evicts each of the dirty sub-blocks in the evicted merged block individually. This can be performed asynchronously with respect to the memory hierarchy operations on the critical path of a memory access.

Conceptually, the STMU performs normal cache operations on sub-blocks. These operations incur additional latencies that add to the existing cache latencies. However, the extra cycles are significantly outweighed by the many lower cache-level/DRAM access cycles saved when using fine-grained accesses to improve cache performance.

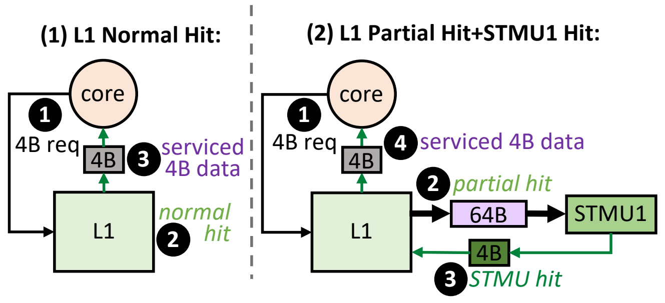


Fig. 10. There are two ways an L1 hit can occur: (1) a request for a non PIA results in a normal hit or (2) a request for a PIA results in a partial hit (base tag match) followed by an STMU hit (sub-tag match).

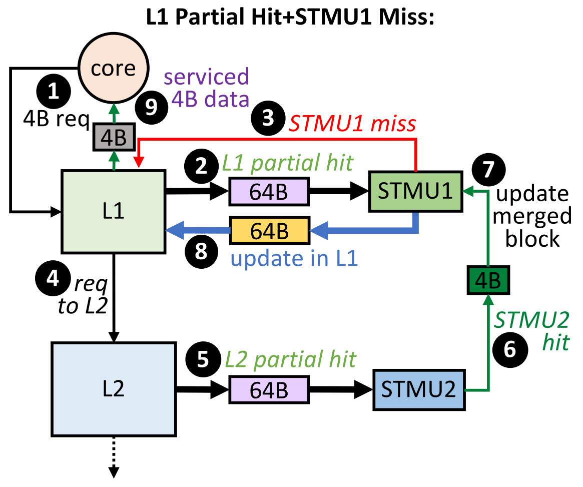


Fig. 11. A request for a PIA can experience a partial hit in the L1 and miss in STMU1. Therefore, the request goes to the L2 and experiences a partial hit (inclusive caches). The request is then followed by an STMU2 hit or miss. If there is a hit, the fetched data is sent to STMU1, which updates the merged block and sends the data to the L1 to be delivered to the core. If there is a miss, then the request is sent to the L3.

5.3 Cache and STMU Interactions

Many different cache and STMU interactions can take place. We describe and illustrate common scenarios that occur.

L1 Hit: Fig. 10 illustrates a normal L1 hit vs. a partial L1 hit that involves the STMU1. *(1) L1 Normal Hit:* A 4B request ❶ experiences a normal L1 hit ❷ because the full tag was present ❸, so the data is sent to the core ❹. This access could not have been a PIA and only incurs the L1 access latency (4 cycles — see Tab. 2).

(2) L1 Partial Hit+STMU1 Hit: A 4B request ❶ experiences a partial L1 hit ❷ because the base tag was present and the full tag was not. The merged block with the partial hit is sent to STMU1, which searches for and locates the sub-tag (hit) ❸ and provides the data to the core ❹. This access was for a PIA and incurs both the L1 access latency and the STMU1 latency (4+2 cycles — see Sec. 5.7).

L1 Partial Hit+STMU1 Miss: Fig. 11 presents scenarios that can result from an STMU1 miss. A 4B request ❶ experiences a partial hit ❷, but experiences an STMU1 miss (sub-tag not present) ❸, triggering an L2 access ❹. This access must be fine-grained due to the partial L1 hit, so it experiences a partial L2 hit, assuming an inclusive cache ❺, and the merged block is sent to STMU2. This access can either hit ❻ or miss in STMU2. The latter results in an access to the L3, which would result in a partial hit and access to

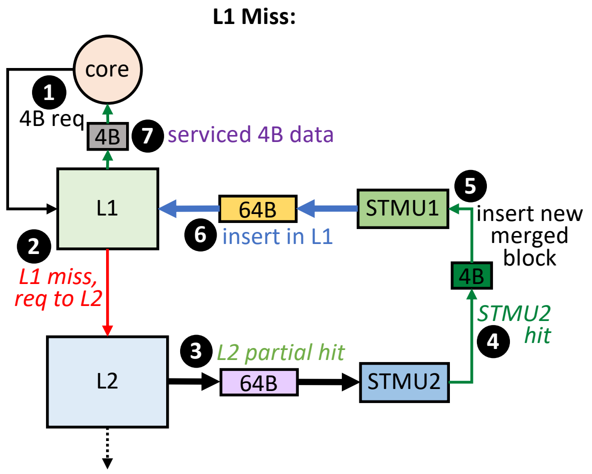


Fig. 12. A request for a PIA results in an L1 miss because the base tag was not present (and consequently neither was the original tag). However, the request experiences a partial hit in the L2, which can be followed by either an STMU2 hit or miss. Following a hit, the requested sub-block data is sent to STMU1, which creates and inserts a new merged block with the new sub-block. This merged block is then inserted into the L1 and the data is delivered to the core.

STMU3. Both scenarios incur the access latencies of the L1 and L2 caches, as well as the latencies of their respective STMUs (4+2+11+3 cycles).

With an STMU2 hit, STMU1 receives the fetched data and updates the merged block that experienced the initial STMU miss ❼. This can result in a sub-block eviction, which would be handled as an update to STMU2. STMU1 then provides the L1 with the updated merged block ❽ and the L1 delivers the requested data to the core ❾. Note that if the STMU2 hit prevents an L2 (and possibly L3) miss due to Graphfire's increased effective L2 capacity, then it saves *29 cycles* (no 34-cycle L3 access) or *229 cycles* (no 34-cycle L3 and 200-cycle DRAM access). The worst-case access latency when Graphfire is enabled corresponds to 258 (4+2+11+3+34+4+200) cycles, where the STMUs are responsible for only 9 (2+3+4). Since Graphfire significantly reduces DRAM accesses, this worst case occurs much less frequently.

L1 Miss: Fig. 12 presents scenarios that can result from a normal L1 miss. A 4B request ❶ experiences an L1 miss (both base and original tag not present), so it accesses the L2 ❷. Either a partial hit ❸, normal hit (before bypassing has been enabled), or L2 miss (both base and original tag not present) followed by an L3 access occurs. If a normal L2 hit occurs, then the data is propagated up the hierarchy and delivered to the core. If a partial L2 hit occurs, the STMU2 access can result in a hit ❹ or miss, which is followed by an L3 request. If an STMU2 hit occurs, STMU1 creates a new merged block (since the base tag was missing in the L1) and inserts the sub-block from STMU2 ❺. This new merged block is then sent to the L1 ❻, which can evict a normal or merged block. If a merged block is evicted, then STMU1 asynchronously handles the eviction of the individual sub-blocks. The L1 delivers the requested data to the core ❼, creating a 20-cycle (4+2+11+3) access. Again, STMU hits prevent memory accesses that would access lower cache levels and potentially DRAM in a conventional hierarchy, shortening many access latencies despite STMU overheads.

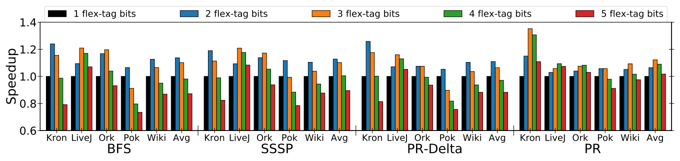


Fig. 13. Speedup comparisons for varying LLC flex-tag bits, normalized to 1 flex-tag bit. On average, 2 bits is most performant; it strikes a balance between mapping flexibility and sufficient sub-block storage.

5.4 Predicting PIA Reuse

Graphfire retains high-reuse PIAs via frequency-based replacement. Thus, each sub-block maintains a frequency count. A normal cache requires additional data to support high frequency counts, but the STMU allows flexible storage within the data block. Thus, additional bits are used for the frequency counts at the expense of a sub-block. Storing one less sub-block per merged block incurs negligible performance loss and 6 frequency bits per sub-block maintains sufficiently high counts to distinguish between PIAs with variable reuse for graphs of varying sizes (Sec. 7.7). At the sub-block level, this incurs no additional hardware overhead, so FBR is used for any sub-block eviction. At the block level, 6-bit counters only require about 1% overhead.

5.5 Flex-Tag Size

The number of flex-tag bits, k, determines the increase in the number of sub-blocks that can share the same block, e.g. $k = 2$ yields $4x$ more mapping flexibility. Furthermore, since each cache level has different index bit mappings, different sub-blocks in each level can compose a merged block, similar to how cache sets contain different normal blocks in each level. Merged blocks in different levels simply have different sub-block metadata, which is handled by the STMUs. Thus, the L1, L2, and L3 values of k are independent of each other; they simply determine the mapping flexibility at each level.

A larger flex-tag provides greater flexibility, but increases sub-tag size, which decreases the number of sub-blocks that can fit in a block. To study this trade-off, we varied the number of flex-tag bits in the LLC (has the greatest impact on performance due to its size). For design simplicity, we maintained the same base tag size across all 3 levels and adjusted k in the L1 and L2 based on the LLC parameter. Fig. 13 presents speedup comparisons of Graphfire operating with different LLC flex-tag sizes, where runtimes are normalized to the configuration with 1 LLC flex-tag bit. 2 flex-tag bits (and therefore 7 in L1 and 4 in L2) is consistently the most performant across multiple application and input combinations (described in Sec. 6), as it strikes an effective balance between mapping flexibility and sub-block storage.

5.6 Cache Coherence

Graphfire maintains coherence at full block granularity, similar to prior works [21], [44]. This design decision avoids modifications to the coherence protocol and maintains the directory's operations for normal blocks. When the directory receives a coherence request for a merged block, it queries the STMUs (corresponding to sharers of the block) to perform sub-block coherence updates. Coherency state bits of sub-blocks are embedded in their metadata, so the STMUs can encapsulate all sub-block coherence operations without impacting the directory capacity or structure. Thus, the only necessary directory modification is the addition of a fine-granularity bit f per cache block, similar to the caches.

Upon a cache miss, the lower level STMU(s) provide the sub-block to insert. The STMU provides the sub-tag, which is appended to the base-tag to form a full tag for directory lookups. STMUs perform fine-grained invalidations/updates if a sub-block's address has been modified by another core. As STMUs perform all operations on sub-blocks, Graphfire does not add complexity to the coherence protocol for managing sub-blocks nor any new messages or states, and does not increase cache coherence traffic.

In-cache directories (even if sparse) pose no issues. Say a core $C2$ reads from a fine-grained address cached by core $C1$ (in S state). To add $C2$ to the sharer's list, its L1 cache controller queries the directory (in L3), sees the full block in S state, and adds $C2$ to the list (using a pointer if directory is sparse). The directory does not know if the address is cached in a normal or merged block in $C2$'s L1. If $C1$ later writes to that address, it experiences a partial hit and its STMU1 upgrades the sub-block coherency state from S to M (metadata update). The directory sends invalidations to all sharers, including $C2$ (as normal). When $C2$'s L1 receives the invalidation, it experiences a partial hit and its STMU1 invalidates the sub-block (metadata update). In the directory, the full block state changes from S to I (as normal).

5.7 STMU Overheads

We used Cacti v7.0 [28] to obtain upper bound estimates for area, power, and timing. Each STMU resembles a very small, *64B total*, fully-associative cache (a mere 0.19% of the 32KB L1 storage). We conservatively modeled 16 ways, each containing a 4B sub-block (Fig. 7) and measure the STMU area to be 0.0065mm^2 (3.3% of the L1 area). Caches modifications to support the STMUs are even more negligible and accounted for in our pessimistic modeling. The dynamic energy per STMU access is 0.0011nJ (4.65% of the L1 energy). Lastly, while the access latency of the L1 is 0.6048ns (0.471ns data + 0.133ns tag), the STMU1 requires 0.187ns. Thus, we conservatively model the STMU1 latency to be 2 cycles (1-cycle access + 1-cycle L1 data update), and the STMU2 and STMU3 latencies as 3 and 4 cycles respectively.

6 Methodology

Applications: This work analyzes three of the most widespread graph processing primitives [8], [10], [38]: (1) **BFS**, (2) **SSSP**, and (3) **PageRank** (listed in Tab. 1). All naturally and efficiently fit the graph processing model (Fig. 1) and thus are bottlenecked by PIAs. Our evaluation uses implementations from the competitive graph DSL, GraphIt [45]. For PageRank, we study both the push-based, work-efficient (PRD) and pull-based, topological implementations (PR).

To measure the predictor's ability to adapt to worklist and working set changes, we also evaluate graph kernels in the context of three multi-phase workloads: (1) **Direction-Optimizing (DO) BFS** [8] is a state-of-the-art BFS implementation that alternates between pull- and push-based phases to minimize the edges traversed in each iteration. (2) **Graph-Sparse** and (3) **Graph-Dense** mimic modern data

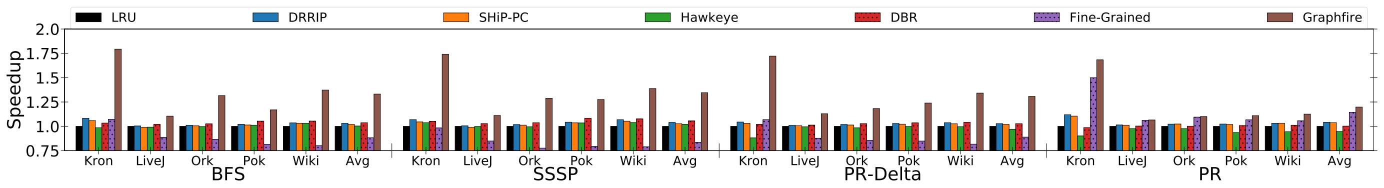


Fig. 14. Speedup comparisons of Graphfire with state-of-the-art and graph-specialized cache management policies. Through automatically identifying PIAs and exploiting synergies between its techniques, Graphfire significantly outperforms all prior cache management policies without relying on software preprocessing.

TABLE 2
OoO vs. In-Order Core Models and Memory Hierarchy Parameters

Parameter	OoO	In-Order
Issue Width	4	1
Instr. Window/ROB/LSQ	128 / 128 / 128	-
Branch Prediction	Gshare [25]	Gshare [25]
Frequency / Tech. Node	2GHz / 22nm	2GHz / 22nm
Area (mm^2)	4.5	0.47
Memory Hierarchy Parameters		
L1	32KB / Private / 8-way / 4-cycle latency	
L2	256KB / Private / 8-way / 11-cycle latency	
L3	single-core: 2MB / Private / 16-way / 34-cycle latency	
	multi-core: 512KB/core / Shared / 16-way / 34-cycle latency	
STMUs	2-cycle (L1), 3-cycle (L2), 4-cycle (L3) latency	
DRAM	DDR3L / 24 GB/s BW / 100ns latency	

analytic workloads, e.g. Graph Neural Networks [40], comprised of graph traversals to gather node/edge features for sparse (SPMV) or dense (SGEMM) matrix processing.

Datasets: Graph application behavior highly depends on input. We select a synthetic power-law Kronecker network that enables quantitative analysis with different data characteristics and also real-world inputs, i.e. social and web networks from LiveJournal, Orkut, Pokec, and Wikipedia, to demonstrate our contributions in a practical setting. The data footprint of each input is much larger ($50-100\times$) than the LLC size. Tab. 1 specifies the inputs and their properties.

Simulator: We utilize MosaicSim [24], a cycle-driven simulator for hardware-software co-design explorations and heterogeneous systems. MosaicSim has been validated against real systems and enables detailed analysis for memory hierarchy tailoring to graph application access patterns. We configured it to use DRAMSim2 [30].

System Parameters: Our evaluation mainly focuses on in-order cores to highlight Graphfire's efficacy and performance achieved even with simple, low-power systems. However, our techniques are core-agnostic and we demonstrate this with a (Haswell-like) out-of-order core model. Tab. 2 presents the two evaluated core models (top) and modeled memory hierarchy (bottom). All evaluated systems have a streaming prefetcher for the L1. In multi-core configurations, the LLC has a static NUCA design with a 512KB local bank per core.

Replacement Policies: We evaluate Graphfire against the following state-of-the-art and domain-specialized policies:

DRRIP [18] targets mixed access patterns, where references can have a near-immediate or distant re-reference interval. DRRIP performs set dueling to apply SRRIP, a scan-resistant policy that prioritizes references with longer or no reuse distances for replacement, or BRRIP, a thrash-resistant policy that aims to keep the working set in the cache.

SHiP [39] learns re-reference intervals of a signature, e.g. PC, memory region, or instruction sequence. By tracking saturating re-reference counters that are updated based on per-signature hits and misses, SHiP learns which signatures have an immediate or distant re-reference interval. We focus on SHiP-PC, which most closely relates to the RRT.

Hawkeye [17] leverages a Belady's algorithm variant to predict whether a LLC reference is cache-friendly or not. By using access pattern history, Hawkeye learns individual reference behaviors to determine which ones benefit from caching. It then prioritizes cache-averse lines for eviction knowing (based on history) which lines' lifetimes overlap.

GRASP [15] introduces domain-specialized cache management for graph analytics. It correlates PIA reuse to vertex degree and leverages RRIP to make replacement decisions. GRASP relies on software reordering to identify groups of high-degree vertices and rearrange them together, while the hardware knows where high-degree vertices are stored. GRASP is implemented on top of Degree-Based Grouping (DBG) [14]. To evaluate hardware-only degree-based replacement **(DBR)**, we model GRASP without software reordering and annotate each PIA with its vertex degree. For each cacheline accessed by PIAs, insertion and hit policies are based on the degree of the vertex *most recently accessed*. In practice, annotations incur additional storage and accesses, but this idealized model evaluates DBR without software reordering.

Fine-Grained We model the baseline with LRU and 4B cachelines to evaluate fine-grained access effects in the absence of Graphfire's other policies.

7 Results

7.1 Application Speedups

Fig. 14 compares runtime performance speedups (normalized to LRU) between Graphfire, DRRIP, SHiP-PC, Hawkeye, and individual replacement and fetch techniques DBR and Fine-Grained. Graphfire outperforms all state-of-the-art techniques, achieving up to a $1.79\times$ speedup (geomean $1.3\times$) over LRU, while prior techniques have negligible improvements. This is because these techniques focus on the binary problem of immediate vs. distant re-reference interval prediction instead of variable fine-grained access reuse. While DRRIP makes replacement decisions based on individual cacheline accesses, it does not learn different access patterns and its operations are too coarse-grained for the PIAs. SHiP-PC learns which PCs are associated with immediate re-reference intervals (PSAs), but treats all PIAs equally because they share the same PC. In fact, SHiP-PC prioritizes them all for eviction due to their irregularity even

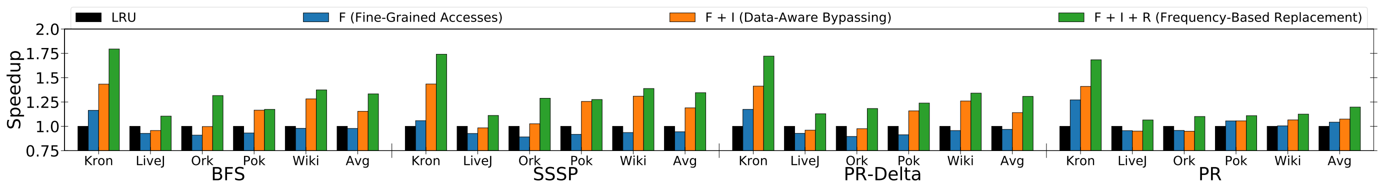


Fig. 15. With synergistic data-aware optimizations—(1) variable (F)etch, (1) data-aware (I)nsertion, and (3) frequency-based (R)eplacement—Graphfire achieves considerable speedups over LRU. Each optimization builds upon the previous and contributes to the total speedup.

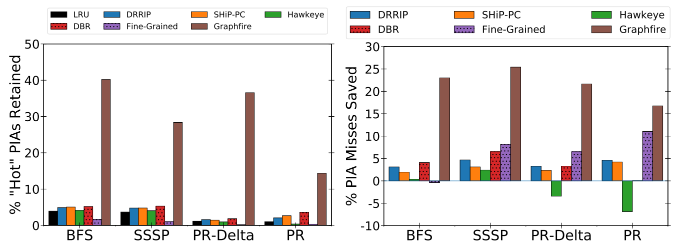


Fig. 16. Comparisons between average percentages of "hot" (high-reuse) vertices retained in the LLC (left) and percentages of PIA misses saved in the LLC (right). Graphfire retains significantly more "hot" vertices and consequently saves significantly more PIA misses compared to prior techniques.

though many of them benefit from being retained in the cache. Hawkeye faces the same problem and requires more hardware to support its history-based learning.

DBR tailors its replacement policy to variable PIA reuse by prioritzing low-degree vertices for eviction, but wastes cacheline space with coarse-grained PIAs and allows all accesses, both streaming and indirect, to thrash in the LLC. The fine-grained baseline also suffers from a lack of data-aware caching, as cache-friendly PSAs experience performance degradation and occupy the LLC, which removes available space for the PIAs. Thus, tailored fetch, insertion, and replacement policies are not effective if applied individually, especially when they are not data-aware. Graphfire attributes its significant performance gains to both *learning* distinct access patterns and optimizing for the PIAs by exploiting *synergies* between its graph-specialized techniques.

7.2 Effects of Graphfire's Cache Policies

Fig. 15 illustrates how Graphfire's composition of tailored fetch, insertion, and replacement policies is key to improving graph analytic performance. First, the STMU enables fine-grained *fetch (F)* for the PIAs to improve cache utilization. Data-aware *insertion (I)* addresses LLC interference by reserving the cache for PIAs. Lastly, frequency-based *replacement (R)* for fine-grained sub-blocks leverages the opportunity (provided by F+I) for per-vertex specialization to accommodate the variable reuse exhibited by PIAs. Each policy builds on the prior for significant performance gains.

Last-Level Cache Utilization: Improved LLC utilization is necessary for Graphfire's performance gains. We measure utilization as the percentage of cacheline *data* accessed before the line is evicted. Graphfire cannot achieve 100% utilization due to metadata storage for the STMU, but achieves 67.18% on average (max. possible is 68.75%). Its techniques together to fit more PIAs in the LLC. Prior works suffer because they achieve only 7.04% utilization on average.

High-Degree PIA Retention: To evaluate Graphfire's ability to retain high-reuse PIAs, we define a "hot" vertex as one of the N most frequently accessed, where N vertices fit in the LLC (524288 4B-vertices fit in 2MB). This quantifies how well Graphfire caches such vertices. Fig. 16 (left) compares the average percentages of "hot" vertices retained. Graphfire's techniques together retain up to 49.6% (avg. 29.9%), outperforming prior works while only using 68.8% of cachelines. This highlights that FBR succeeds when specialized fetch and insertion increase the effective LLC size.

PIA Miss Rate Reduction: Graphfire's primary goal is to improve performance through PIA specialization. Fig. 16 (right) highlights that Graphfire on average reduces 21.7% of PIA misses in the LLC, while prior works are not nearly as successful. Graphfire thus achieves up to a $3.92\times$ reduction (geomean $1.83\times$) in DRAM accesses, which improves memory bandwidth efficiency by up to $2.28\times$ (geomean $1.41\times$). Graphfire can synergize with aggressive latency tolerance approaches, e.g. prefetching, by alleviating bandwidth consumption to provide additional performance improvements.

7.3 Scalability

When bandwidth is limited in a multi-core system, many latency tolerance mechanisms suffer. Fig. 17 compares the average bandwidth usages (GB/s) (top) and performance speedups (bottom) of Graphfire and LRU on the Kronecker network (we focus on this input to avoid redundancy) when scaling from 1-64 cores, normalized to LRU. With more cores, both techniques exploit more memory-level parallelism (MLP), consuming more bandwidth.

While both techniques demonstrate performance improvements due to MLP, LRU nears the memory bandwidth limit of 24 GB/s at 64 cores, limiting its scalability, while Graphfire reduces DRAM accesses to be more bandwidth-efficient. Graphfire not only consistently outperforms LRU despite synchronization overheads, but also scales with more parallelism. With 64 cores, it achieves up to a $71.3\times$ speedup (geomean $63.3\times$) over the single-thread baseline, while LRU achieves up to $64.9\times$ (geomean $47\times$).

7.4 Software Reordering

Software reordering aims to exploit structural properties of graphs to improve locality. Sophisticated techniques require many application executions to amortize preprocessing costs [5]. Even lightweight techniques, e.g. DBG [14], are less effective when a fraction of the graph is traversed. Search algorithms, e.g. BFS, traverse the graph until they find desired data. In such cases, reordering can be costly, especially if the graph is very large. Graphfire as a hardware-based alternative avoids such preprocessing costs.

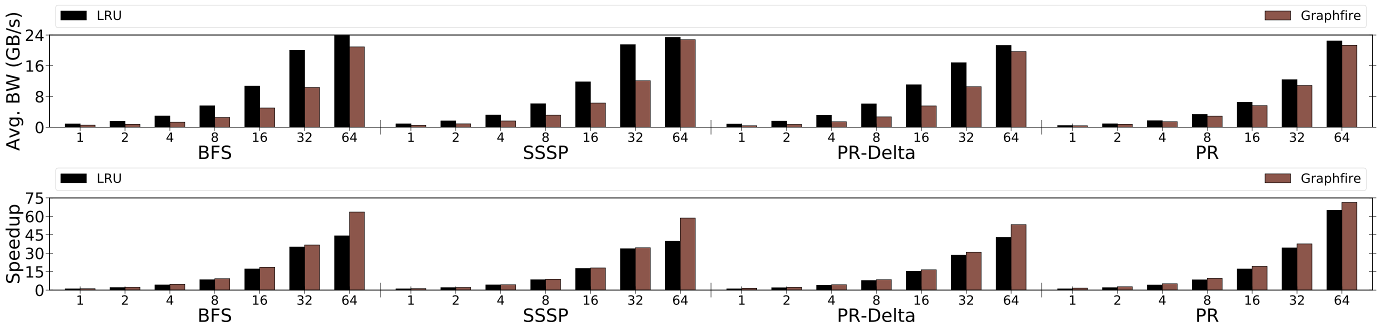


Fig. 17. Average BW (top) and speedup (bottom) comparisons between Graphfire and LRU when scaling from 1 to 64 cores on the Kron input. All speedups are normalized to single-thread LRU performance. As LRU reaches the BW limit, it stops reaping benefits from memory-level parallelism and its scalability suffers. Meanwhile, Graphfire performance gains scale due to its improved bandwidth efficiency, offering more improvements with more parallelism.

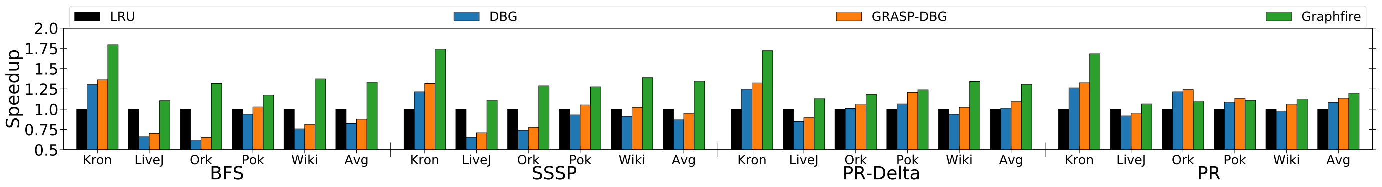


Fig. 18. Speedup comparisons between Graphfire, DBG software reordering, and GRASP. The densest frontier is traversed and all speedups are normalized to LRU. Graphfire consistently outperforms all techniques, as reordering costs cannot always be amortized, especially when only part of the graph is traversed.

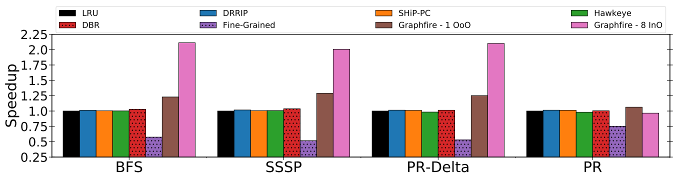


Fig. 19. Equal area geomean speedup comparisons of Graphfire (OoO and in-order) with state-of-the-art replacement policies on OoO. Graphfire consistently outperforms prior works on an OoO core and is most performant with in-order cores on push-based applications.

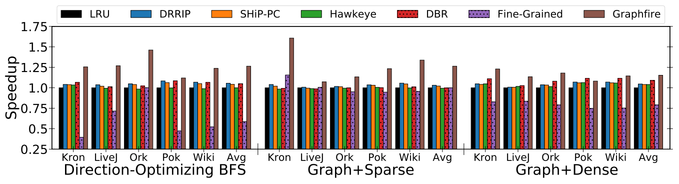


Fig. 20. Speedup comparisons of Graphfire with state-of-the-art on multi-phase applications. Graphfire accelerates sparse phases without degrading dense phases, yielding significant speedups overall.

Fig. 18 compares Graphfire to DBG software reordering and GRASP, whose replacement policy heavily relies on DBG. All runtimes were measured when the application runs on its densest and most representative frontier, with performances normalized to LRU. When factoring in preprocessing costs, Graphfire outperforms both reordering-based techniques, achieving up to a 1.52× speedup (geomean 1.21×) over DBG, while GRASP does not offer significant improvements over DBG. In some cases, reordering even hurts performance. Graphfire's locality predictor identifies PIAs with 100% accuracy even with a reordered graph, as no reordering scheme can consistently utilize at least 50% of cachelines similar to streaming contiguous PIAs. However, Graphfire's cache policies are designed for PIAs with poor locality, so it does not make sense to incur the cost of reordering and use Graphfire.

7.5 Adaptability

Graphfire's techniques are agnostic to the core model. Fig. 19 shows that Graphfire offers significant speedups over prior works in an OoO setting (solid colored bars), achieving up to 1.6× speedup (geomean 1.2×). Graphfire offers improvements on top of OoO core structures, e.g. wide issue queues, ROBs, and LSQs, but as pointed in [6], these structures are underutilized when executing graph applications. To explore a more area-efficient configuration, we trade these structures for greater MLP and perform an equal-area speedup comparison between 8 in-order cores and 1 OoO (Tab. II). Thus, Fig. 19 compares to Graphfire running on 8 in-order cores with all runtimes normalized to 1 OoO with LRU. The in-order configuration is the most performant, achieving up to a 2.45× speedup (geomean 1.71×).

General-Purpose Workload Performance: Graphfire targets sparse and irregular graph analytic workloads that *lack spatial locality*. Therefore, its techniques are specifically designed for fine-grained memory accesses and naturally are not expected to be amenable to traditional, cache-friendly applications. We evaluate our approach with the entire Parboil benchmark suite [34], SPEC2006 workloads, and additional sparse linear algebra applications SPMM and Sparse-Dense Hadamard Product (SDHP) [36]. Four workloads, MRI-Q, SAD, SGEMM, and TPACF, experience no performance effects and three workloads, CUTCP, SPMM, and SDHP, experience $< 10\%$ slowdowns relative to LRU. The remaining workloads experience 22-88% performance slowdowns. These slowdowns arise because the RRT is not currently designed to recognize all (cache-friendly) memory access patterns, though it could be combined with prefetchers to become more robust. For applications with varying amounts of cache-friendly locality, Graphfire should simply be disabled as described in Sec. 3.1, or their performances can suffer.

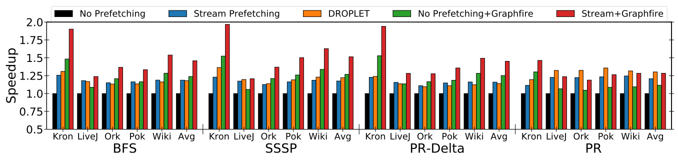


Fig. 21. Speedup comparisons of Graphfire with graph-specialized prefetcher DROPLET (all on an OoO core). Graphfire significantly outperforms DROPLET, which has low prefetch accuracy for PSAs used to prefetch PIAs.

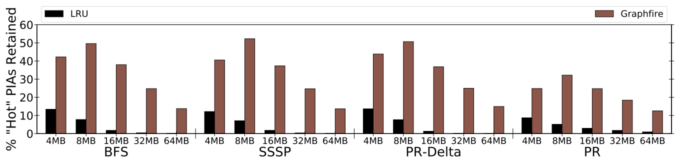


Fig. 22. Comparisons between "hot" PIA retention percentages in the LLC for Kron networks of varying sizes (x-axis shows PIA footprint, which is always larger than LLC capacity). Graphfire always caches hot vertices better.

Multi-Phase Adaptation: Graphfire still retains high performance for many dense workloads at low area, unlike prior graph acceleration work. This is useful for graph analytic kernels that involve dense computations, e.g. over vertices. To evaluate the RRT's robustness and adaptability in these scenarios, Fig. 20 compares Graphfire to prior techniques on multi-phase applications DO BFS, Graph+Sparse, and Graph+Dense. Graphfire outperforms all prior techniques across the board, achieving up to a $1.61\times$ speedup (geomean $1.22\times$). DO BFS begins and ends with work-efficient push-based phases, but primarily has bandwidth-efficient pull-based phases that exhibit more locality. Graph+Sparse and Graph+Dense alternate between graph (SSSP) and matrix (SPMV/SGEMM) computations on a per-vertex basis. The RRT identifies all primary access patterns accurately and quickly adapts to worklist and working set changes. Graphfire targets graph phases (and improves sparse phase cache utilization) without harming dense phases, ultimately benefiting multi-phase and co-located workloads.

7.6 Prefetching

Pattern-based prefetchers for temporal access sequences and spatial locality prediction [11], [20] do not target PIAs. Recent work explored pointer-based prefetching for irregular accesses [32], [42], but these works struggle with variable indirect access patterns in graph applications or require significant training. Even graph-tailored prefetchers [2], [3], [6] struggle with timeliness and accuracy due to varying amounts of computation and control flow.

Fig. 21 compares Graphfire (with and without the baseline L1 stream prefetcher) to **DROPLET** [6] a state-of-the-art data-aware decoupled prefetcher for graph analytics. By using an L2 streaming prefetcher for PSAs, it aims to fetch PIAs early. Because DROPLET's design is informed by OoO execution characterizations, we measure all runtimes using 1 OoO and normalize to *no* prefetching. Both Graphfire configurations consistently outperform DROPLET, which averages 63.3% streaming access prefetch accuracy that harms PIA prefetch accuracy. Prefetching does not reduce DRAM access frequency. In contrast, Graphfire reduces the frequency and latency of PIAs without requiring invasive hardware specializations, e.g. to support address calculation and address snooping modules, and prefetch buffers. These overheads far surpass Graphfire's, e.g. [3] uses full in-order cores (orders of magnitude larger than STMUs) as programmable prefetchers.

7.7 Sensitivity to Graph Size

To evaluate Graphfire's performance on varying graph sizes, Fig. 22 compares the percentages of "hot" vertices retained by LRU and Graphfire as Kronecker networks grow in size from $2^{20} - 2^{24}$ nodes (400MB - 7GB) whereas the LLC always stores 2MB. Graphfire consistently demonstrates significant improvements over LRU and all state-of-the-art approaches, which perform similarly (not shown due to space). For *all* application/dataset pairs, Graphfire advances state-of-the-art and yields speedups ranging from $1.11 - 1.79\times$. These speedups attribute to Graphfires improvement of cache utilization, which fits many more PIAs in the LLC, lowers the probability of PIAs thrashing the cache, and retains a much greater fraction of the hot PIAs regardless of the graph size.

With larger graphs, it is more difficult for counters to learn access frequencies. Massive graphs are very challenging for any caching technique; e.g. using a 2MB LLC in this experiment to cache 64MB of PIAs (from a $\sim$7GB graph) leads to thrashing amongst the PIAs alone. However, Graphfire's alleviates thrashing effects with more success than state-of-the-art caching policies. Multi-core architectures are most suitable for such large graphs where the reuse distances between vertices consistently exceeds the cache capacity. With these systems, Graphfire can leverage their aggregate LLC capacities to yield even more improvement.

8 Discussion

Adaptive Cache Lines: In other domains, accesses may benefit from varying sub-block sizes. The STMU's flexible design can manage sub-block sizes beyond 8B via a "granularity" bit in each merged block's metadata (the g bit in Fig. 7-9). By allocating more than 1 bit to this field, the STMU can support more sub-block sizes, e.g. {4B, 8B, 16B, 32B}.

Prefetch-Aware Caching: Graphfire offers opportunities for additional performance improvements via prefetching. For example, an indirect memory access prefetcher, e.g. IMP [42], can interface with the STMU to cache more prefetches and consequently eliminate more PIA cold misses. Domain-specific prefetchers can also synergize with the RRT to identify spatial locality *non-speculatively* and further enhance locality prediction.

9 Related Work

Cache Partitioning: Many techniques aim to improve cache utilization by reserving different parts of the cache for different threads, cores, or applications [9], [12], which is beneficial for simultaneous workload execution. Graphfire instead tailors its insertion scheme at a finer granularity to focus on heterogeneous access patterns within a single graph application. XMem [37] dedicates partial or full cache areas to blocks with high-reuse so they cannot be evicted by other accesses. This is only practical when all high-reuse blocks fit in the cache. Large graphs with thousands of high-degree vertices require more than the LLC capacity.

Partial Cachelines: Several works tailor to spatial locality [21], [44] by offering complex designs or high metadata overheads to support a wide range of cacheline sizes. Tag-split cache [23] divides tags to support coarse and fine-grained accesses for GPGPUs, but incurs area overhead to store partial cacheline tags. Decoupled sector caches [31] share tags between cachelines from different sectors. Graphfire avoids tag sharing with hierarchical tags. IMP [42] focuses on PIAs, but requires partial cachelines to remain continuous, limitating their performance gains.

Cache Replacement Policies: Sophisticated variants of LRU, LFU, RRIP, and other cache efficiency metrics [18], [29], [43] rely on static metrics that do not adapt well to varying, distinct access patterns and especially the variable reuse of PIAs in graph analytics. More recent approaches aim to *learn* varying access patterns through history-based techniques or via machine learning [19], [33], [35], which can require long, computationally intensive offline training. Unfortunately, even intelligent learning schemes can be ineffective for graph applications as PIAs rarely, if ever, exhibit patterns to learn, making such policies potentially costly. Graphfire instead exploits synergies between PIA-tailored policies to non-speculatively focus on problematic accesses.

P-OPT [4] proposes an architecture solution that uses the transpose of a graph's adjacency matrix, referred to as the Rereference Matrix, to implement Belady's MIN replacement policy. Constructing the Rereference Matrix incurs a preprocessing cost, while Graphfire remains software-agnostic. Furthermore, Fig. 14 and 15 demonstrate that replacement alone offers limited speedup, as even an optimal policy cannot prevent high-reuse PIAs from being evicted if too many (more than the associativity) map to the same set. Improving cache utilization is key to creating more performance opportunity for tailored replacement.

Graph-Tailored Memory Hierarchies: Domain-specific memory subsystem designs augment hardware with specialized engines or accelerators. These modules perform computations or alter traversal scheduling to exploit power-law properties or graph locality [1], [26], [27]. As a result, such approaches require significant hardware modifications within the caches themselves, as well as ISA changes, which are difficult to implement in general-purpose systems.

10 Conclusion

This work presents Graphfire, a flexible, fully hardware-based memory hierarchy approach that learns and optimizes for irregular accesses in graph applications. By leveraging the key observations that PIAs require specialized treatment, can be identified in hardware, and have a subset that benefit from caching, Graphfire learns distinct memory access patterns on a per-PC basis to identify PIAs and synergizes data-aware fetch, insertion, and replacement policies tailored to them. This results in considerable performance improvements and bandwidth efficiency over state-of-the-art techniques, allowing graph applications to scale on general-purpose, multi-core, shared-memory systems, where software can run quickly and unchanged. This work is a timely contribution for big data processing, as graph algorithms at the heart of data analytics must keep up with ever-growing modern network trends.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback. This research was supported in part by the DARPA SDH Program under agreement No. FA8650-18-2-7862 and NSF Award No. 1763838. Aninda Manocha was supported by the NSF Graduate Research Fellowship. Prof. Aragón was supported by the Spanish State Research Agency under grant TIN2016-75344-R and by Fundación Séneca-Agencia de Ciencia y Tecnología, Región de Murcia, Programa Jiménez de la Espada (21508/EE/21). The authors' views and conclusions contained herein should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA or NSF.

References

[1] A. Addisie, H. Kassa, O. Matthews, and V. Bertacco, "Heterogeneous memory subsystem for natural graph analytics," in *The 2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Sep. 2018, pp. 134–145.

[2] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *The International Conference on Supercomputing (ICS)*. ACM, 2016.

[3] ——, "An event-triggered programmable prefetcher for irregular workloads," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2018, pp. 578–592.

[4] V. Balaji, N. Crago, A. Jaleel, and B. Lucia, "Practical optimal cache replacement for graphs," in *Proceedings of the 2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2021.

[5] V. Balaji and B. Lucia, "When is graph reordering an optimization? Studying the effect of lightweight graph reordering across applications and input graphs," in *The 2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 203–214.

[6] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 373–386.

[7] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an Ivy Bridge server," in *The 2015 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2015.

[8] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Press, 2012, pp. 1–10.

[9] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 213–224.

[10] V. T. Chakaravarthy, F. Checconi, P. Murali, F. Petrini, and Y. Sabharwal, "Scalable single source shortest path algorithms for massively parallel systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 2031–2045, July 2017.

[11] C. F. Chen, S. . Yang, B. Falsafi, and A. Moshovos, "Accurate and complexity-effective spatial pattern prediction," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA)*, 2004, pp. 276–287.

[12] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *Proceedings of 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 104–117.

[13] G. Erkan and D. R. Radev, "LexRank: Graph-based lexical centrality as salience in text summarization," *Journal of Artificial Intelligence Research (JAIR)*, vol. 22, pp. 457–479, 2004.

[14] P. Faldu, J. Diamond, and B. Grot, "A closer look at lightweight graph reordering," in *The 2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Press, 2019.

[15] ——, "Domain-specialized cache management for graph analytics," in *Proceedings of the 2020 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2020.

[16] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," in *Proceedings of the 10th Annual International Symposium on Computer Architecture (ISCA)*, 1983, pp. 124–131.
[17] A. Jain and C. Lin, "Back to the future: Leveraging Belady's algorithm for improved cache replacement," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. IEEE Press, 2016, p. 78–89.
[18] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2010, pp. 60–71.
[19] A. V. Jamet, L. Alvarez, D. A. Jiménez, and M. Casas, "Characterizing the impact of last-level cache replacement policies on big-data workloads," in *The 2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 134–144.
[20] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI. ACM, 1990, pp. 364–373.
[21] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, "Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy," in *Proceedings of the 45th Annual International Symposium on Microarchitecture (MICRO)*, 2012, p. 376–388.
[22] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *Journal of Machine Learning Reseach (JMLR)*, vol. 11, pp. 985–1042, Mar. 2010.
[23] L. Li, A. B. Hayes, S. L. Song, and E. Z. Zhang, "Tag-split cache for efficient GPGPU cache utilization," in *Proceedings of the 2016 International Conference on Supercomputing (ICS)*. ACM, 2016.
[24] O. Matthews, A. Manocha, D. Giri, M. Orenes Vera, E. Tureci, T. Sorensen, T. J. Ham, J. L. Aragón, L. Carloni, and M. Martonosi, "MosaicSim: A lightweight, modular simulator for heterogeneous systems," in *Proceedings of the 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.
[25] S. Mcfarling, "Combining branch predictors," Digital Western Laboratory, Tech. Rep. WRL TN-36, 1993.
[26] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling," in *The International symposium on Microarchitecture (MICRO)*, October 2018.
[27] A. Mukkara, N. Beckmann, and D. Sanchez, "PHI: Architectural support for synchronization- and bandwidth-efficient commutative scatter updates," in *Proceedings of the 52nd Annual International Symposium on Microarchitecture (MICRO)*, 2019, pp. 1009–1022.
[28] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," *HP laboratories*, vol. 27, pp. 1–24, 2009.
[29] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. ACM, 1990, pp. 134–142.
[30] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.
[31] A. Seznec, "Decoupled sectored caches: Conciliating low tag implementation cost and low miss ratio," in *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, 1994, pp. 384–393.
[32] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.
[33] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual International Symposium on Microarchitecture (MICRO)*, 2019, pp. 413–425.
[34] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing,"
[35] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.
University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, 2012.
[36] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688
[37] N. Vijaykumar, A. Jain, D. Majumdar, K. Hsieh, G. Pekhimenko, E. Ebrahimi, N. Hajinazar, P. B. Gibbons, and O. Mutlu, "A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory," in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 207–220.
[38] J. J. Whang, A. Lenharth, I. S. Dhillon, and K. Pingali, "Scalable data-driven PageRank: Algorithms, system issues, and lessons learned," in *The European Conference on Parallel Processing (Euro-Par)*, 2015, pp. 438–450.
[39] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual International Symposium on Microarchitecture (MICRO)*, 2011, pp. 430–441.
[40] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Networks Learn. Syst.*, vol. 32, no. 1, pp. 4–24, 2021.
[41] W. Xiao, J. Xue, Y. Miao, Z. Li, C. Chen, M. Wu, W. Li, and L. Zhou, "Tux^2: Distributed graph computation for machine learning," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 669–682.
[42] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in *Proceedings of the 48th Annual International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 178–190.
[43] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the memory system: Predicting and optimizing memory behavior," in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, May 2002, pp. 209–220.
[44] C. Zhang, Y. Zeng, and X. Guo, "Scrabble: A fine-grained cache with adaptive merged block," *IEEE Transactions on Computers*, vol. 69, no. 1, pp. 112–125, 2020.
[45] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "GraphIt: A high-performance graph DSL," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 2, no. OOPSLA, pp. 1–30, Oct. 2018.

Aninda Manocha is currently a Computer Science PhD student at Princeton University advised by Margaret Martonosi. Her broad area of research is computer architecture, with specific interests in data supply techniques across the computing stack for graph and other emerging applications with sparse memory access patterns. She received her B.S. degrees in Electrical and Computer Engineering and Computer Science from Duke University in 2018.

Juan L. Aragón is an Associate Professor in Computer Architecture at the University of Murcia (UMU), Spain. He received his Ph.D. degree in Computer Engineering in 2003 from UMU, followed by a 1-year postdoc at University of California, Irvine. He has been a Visiting Researcher at EPFL and Princeton University. He has coauthored +55 research papers in major conferences and journals. His research focuses on computer architecture, heterogeneous systems, application-specific accelerators and GPUs.

Margaret Martonosi is the Hugh Trumbull Adams '35 Professor of Computer Science at Princeton University. Her research focuses on computer architecture and hardware–software interface issues in both classical and quantum systems. Her work has included the widely-used Wattch power modeling tool and the Princeton ZebraNet mobile sensor network project. Martonosi received her Ph.D. degree in Electrical Engineering from Stanford University. She is a Fellow of IEEE and ACM.